Stream Types

JOSEPH W. CUTLER, University of Pennsylvania, USA
CHRISTOPHER WATSON, University of Pennsylvania, USA
EMEKA NKURUMEH, California Institute of Technology, USA
PHILLIP HILLIARD, University of Pennsylvania, USA
HARRISON GOLDSTEIN, University of Pennsylvania, USA
CALEB STANFORD, University of California, Davis, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA

We propose a rich foundational theory of typed data streams and stream transformers, motivated by two high-level goals. First, the type of a stream should be able to express complex *sequential patterns* of events over time. And second, it should describe the internal *parallel structure* of the stream, to support deterministic stream processing on parallel and distributed systems. To these ends, we introduce *stream types*, with operators capturing sequential composition, parallel composition, and iteration, plus a core calculus λ^{ST} of *transformers* over typed streams that naturally supports a number of common streaming idioms, including punctuation, windowing, and parallel partitioning, as first-class constructions. λ^{ST} exploits a Curry-Howard-like correspondence with an ordered variant of the Logic of Bunched Implication to program with streams compositionally and uses Brzozowski-style derivatives to enable an incremental, prefix-based operational semantics. To illustrate the programming style supported by the rich types of λ^{ST}, we present a number of examples written in Delta, a prototype high-level language design based on λ^{ST}.

CCS Concepts: • **Software and its engineering → Specialized application languages**; • **Theory of computation → Type structures**.

Additional Key Words and Phrases: Type Systems, Stream Processing, Ordered Logic, Bunched Implication

ACM Reference Format:
Joseph W. Cutler, Christopher Watson, Emeka Nkurumeh, Phillip Hilliard, Harrison Goldstein, Caleb Stanford, and Benjamin C. Pierce. 2024. Stream Types. *Proc. ACM Program. Lang.* 8, PLDI, Article 204 (June 2024), 25 pages. https://doi.org/10.1145/3656434

1 INTRODUCTION

What is the type of a stream? A straightforward answer, dating back to the early days of functional programming [17], is that a stream is an unbounded sequence of items drawn from a single fixed type, produced by one part of a system (or the external world) and consumed by another. This simple perspective has been immensely successful and can be found in the programming models exposed by today's most popular distributed stream processing eDSLs (e.g., Flink [18, 30], Beam [35], Storm [34], and Heron [31]).

Authors' addresses: Joseph W. Cutler, jwc@seas.upenn.edu, University of Pennsylvania, USA; Christopher Watson, University of Pennsylvania, USA; Emeka Nkurumeh, California Institute of Technology, USA; Phillip Hilliard, University of Pennsylvania, USA; Harrison Goldstein, University of Pennsylvania, USA; Caleb Stanford, University of California, Davis, USA; Benjamin C. Pierce, University of Pennsylvania, USA.

ACM 2475-1421/2024/6-ART204
https://doi.org/10.1145/3656434

However, this homogeneous treatment of streams leaves something to be desired. For one thing, temporal patterns like *bracketedness* (every "begin" event has a following "end") or the expectation that exactly k events will arrive on a given stream are invisible in its type. Programmers get no help from the type system to ensure such properties when producing a stream, nor can they rely on them when consuming one.

Another issue with the homogeneous stream abstraction is that streaming data sometimes arrives at a processing node from multiple sources *in parallel*. Processing these as a single homogenous stream often involves using arrival times to impose an "incidental" order on parallel data. This can make it difficult to ensure that processing is deterministic, because downstream results may then depend on factors like network latency [59, 67].

Our principal contribution is a novel logical foundation for typed stream processing that can precisely describe streams with both complex sequential patterns and parallel structure. On this foundation, we build a calculus called λ^{ST} that is (a) expressive and type-safe for streams with such complex temporal patterns and (b) deterministic, even when inputs can arrive from multiple sources in parallel. We also present Delta, an experimental language design based on λ^{ST} with a simple sequential interpreter (a full-blown distributed implementation is left for future work).

Programs in λ^{ST} are intuitively batch processors that operate over entire streams "in one gulp." But, since streams are in general unbounded, stream transformers can't actually wait for the entire input stream to arrive before producing any output. The operational semantics of the λ^{ST} calculus is therefore designed to be *incremental*, producing partial outputs from partial inputs on the fly. A λ^{ST} program is interpreted as a function mapping any prefix of its input(s) to a prefix of its output plus a "derivative" term that is ready to transform the rest of the input to the rest of the output.

Our stream types include two kinds of products, one representing a pair of streams in temporal sequence, the other a pair of streams in parallel. This structure is inspired both by Concurrent Kleene Algebras [43, 52], which describe partially ordered series/parallel data, and by work by Alur et al. [3] and Mamouras et al. [59] in which streams are modeled as partially ordered sets. We find a suitable proof theory for this two-product formalism in a variant of O'Hearn and Pym's Logic of Bunched Implications (BI) [63]. BI is well known as a foundation for separation logic [66], where its "separating conjunction" allows for local reasoning about separate regions of the heap in imperative programs. In λ^{ST}, we replace *spatial* with *temporal* separation: one of our products describes pairs of streams separated sequentially in time; the other describes pairs of temporally independent streams whose elements may arrive in interleaved fashion.

Concretely, our contributions are:

(1) We propose *stream types*, a static discipline for distributed stream processing that generalizes the traditional homogeneous view of streams to a richer nested-parallel-and-sequential structure, and λ^{ST}, a calculus of stream processing transformers inspired by a Curry-Howard-like correspondence with an ordered variant of BI. Terms in λ^{ST} are high-level programs in a functional style that conceptually transform whole streams at once.
(2) We equip λ^{ST} with an operational semantics interpreting terms as incremental transformers that accept and produce finite prefixes of streams. Our main result is a powerful *homomorphism theorem* (Theorem 3.2) guaranteeing that the result of a transformer does not depend on how the input stream is divided into prefixes. This theorem implies that the semantics is deterministic: all interleavings of parallel sub-streams yield the same final result.
(3) We present Delta, an experimental high-level functional language prototype based on the λ^{ST} calculus that serves as a tool for exploring the potential of richly typed stream programming. We demonstrate by example how Delta enables type-safe programming over streams with complex patterns of parallel and sequential data and how it prevents nondeterminism.

Common programming patterns from stream processing practice that are elegantly supported by this richer model include MapReduce-like pipelines, temporal integrity constraints, windowing, punctuation, parallelism control, and routing.

Section 2 below explores some concrete cases where λ^{ST}'s structured types can prevent common stream processing bugs and enable cleaner programming patterns. Section 3 presents Kernel λ^{ST}, a minimal subset with just the features needed to state and understand the main technical results. Section 4 extends this presentation to Full λ^{ST}. Section 5 develops several further examples. Sections 6 and 7 discuss related and future work. An overview of our prototype implementation of Delta, technical details omitted from the main paper, and code for the examples can all be found in the extended version [22].

2 MOTIVATING EXAMPLES

Types for temporal invariants. Consider a stream of brightness data coming from a motion sensor, where each event in the stream is a number between 0 and 100. Suppose we want a stream transformer that acts as a thresholded filter, sending out a "`Start`" event when the brightness level goes above 50, forwarding along brightness values until the level dips below the threshold, and sending a final "`Stop`" event. For example:

$$11, 30, 52, 56, 53, 30, 10, 60, 10, \ldots \implies \texttt{Start}, 52, 56, 53, \texttt{Stop}, \texttt{Start}, 60, \texttt{Stop}, \ldots$$

The output of the transformer should satisfy the following *temporal invariant*: each start event must be followed by one or more data events and then one end event. Conventional stream processing systems would give this transformation a type like `Stream Int` $\longrightarrow$ `Stream (Start + Int + Stop)`, which expresses only the possible kinds of events in the output, not the temporal invariant that the `Start` must come before the numbers and the `Stop` after.

These simple types are even more problematic when *consuming* streams. Suppose another transformer wants to consume the output stream of type `Stream (Start + Int + Stop)` and compute the average brightness between each start/end pair. We know *a priori* that the stream is well bracketed, but the type does not say so. Thus, the second transformer must *re-parse* the stream to compute the averages, requiring additional logic for various special cases (`Stop` before `Start`, empty `Start`/`Stop` pairs) that cannot actually occur in the stream it will see.

In λ^{ST}, we can express the required invariant with the type $(\texttt{Start} \cdot \texttt{Int} \cdot \texttt{Int}^\star \cdot \texttt{End})^\star$, specifying that the stream consists of a start message, at least one `Int`, and an end message, repeatedly. A well-typed transformer with this output type is guaranteed to enforce this invariant, and, conversely, a downstream transformer can assume that its input will adhere to it.

Enforcing deterministic parallelism. A second limitation of homogeneous streams is that they impose a *total ordering* on their component events. In other words, for each pair of events in the stream, a transformer can tell which came first. This is problematic in a world where stream transformers work over data that is logically only partially ordered—e.g., because it comes from separate sources.[1]

For example, consider a system with two sensors, each producing one reading per second and sending these readings via different network connections to a single transformer that averages them pairwise, producing a composite reading each second. A natural way to do this is to merge the two streams into a single one, group adjacent pairs of elements (i.e., impose a size-two tumbling window), and average the pairs. But this is subtly wrong: a network delay could cause a pair of

[1]The same objection applies for stream processing systems that impose total *per key* ordering of a parallelized stream—cf. `KeyedStream` in Flink [30]—since data associated with a given key may also come from multiple sources in parallel.

consecutive elements in the merged stream to come from the same sensor, after which the averages will all be bogus.

The problem with this transformer is that it is not deterministic: its result can depend on external factors like network latency. Bugs of this type can easily occur in practice [59, 67] and can be very difficult to track down, since they may only manifest under rare conditions [50].

Once again, this is a failure of type structure. In λ^{ST}, we can prevent it by giving the merged stream the type $(\texttt{Sensor1}\|\texttt{Sensor2})^\star$, capturing the fact that it is a stream of *parallel pairs* of readings from the two sensors. We can write a strongly typed merge operator that produces this type, given parallel streams of type $\texttt{Sensor1}^\star$ and $\texttt{Sensor2}^\star$. This merge operator is deterministic—indeed, *all* well-typed λ^{ST} programs are, as we show in Section 3.3). Operationally, it waits for events to arrive on both of its input streams before sending them along as a pair.

3 KERNEL λ^{ST}

In this section, we define the most important constructors of stream types and the corresponding features of the term language; these form the "kernel" of the λ^{ST} calculus. The rest of the types and terms of Full λ^{ST} will layered on bit by bit in Section 4.

The *concatenation* constructor $\cdot$ describes streams that *vary* over time: if s and t are stream types, then $s \cdot t$ describes a stream on which all the elements of s arrive first, followed by the elements of t. A producer of a stream of type $s \cdot t$ must first produce a stream of type s and then a stream of type t, while a consumer can assume that the incoming data will first consist of data of type s and then of type t. The transition point between the s and t parts is handled automatically by λ^{ST}'s semantics: the underlying data of a stream of type $s \cdot t$ includes a *punctuation marker* [75] indicating the cross-over. One consequence of this is that, unlike Kleene Star for regular languages, streams of type $s^\star$ are distinguishable from streams of type $s^\star \cdot s^\star$ because a transformer accepting the latter can see when its input crosses from the first $s^\star$ to the second.

On the other hand, the *parallel* stream type $s\|t$ describes a stream with two parallel substreams of types s and t. Semantically, the s and t components are produced and consumed independently: a transformer that produces $s\|t$ may send out an entire s first and then a t, or an entire t and then the s, or any interleaving of the two. Conversely, a transformer that accepts $s\|t$ must handle all these possibilities uniformly, by processing the s and t parts independently. To enable this, each element in the parallel stream is tagged to indicate which substream it belongs to. This means that streams of type $s\|t$ are isomorphic, but not identical, to streams of type $t\|s$, and similarly $\texttt{Int}^\star\|\texttt{Int}^\star$ is not the same as $\texttt{Int}^\star$.

Parallel types can be combined with concatenation types in interesting ways. For example, a stream of type $(s\|t) \cdot r$ consists of a stream of interleaved items from s and t, followed (once all the s data and t data has arrived) by a stream of type r. By contrast, a stream of type $(s \cdot t)\|(s' \cdot t')$ has two interleaved components, one a stream described by s followed by a stream described by t and the other an s' followed by a t'. The fact that the parallel type is on the outside means that the change-over points from s to t and s' to t' are completely independent.

The base type 1 describes a stream containing just one data item, itself a unit value. The other base type is ε, the type of the empty stream containing no data; it is the unit for both the $\cdot$ and $\|$ constructors—i.e., $s \cdot \varepsilon$, $\varepsilon \cdot s$, $\varepsilon\|s$ and $s\|\varepsilon$ are all isomorphic to s, in the sense that there are λ^{ST} transformers that convert between them.

In summary, the Kernel λ^{ST} stream types are given by the grammar on the top left in Figure 1. (So far, these types can only describe streams of fixed, finite size. In Section 4.2 we will enrich the kernel type system with unbounded streams via the Kleene star type $s^\star$.)

What about terms? Recall that our goal is to develop a language of core terms e, typed by stream types, where well-typed terms $x : s \vdash e : t$ are interpreted as stream transformers accepting a

stream described by s and producing one described by t. The term e runs by accepting some inputs as described by s, producing some outputs as described by t, and then stepping to a new term e', with an updated type t', that is ready to accept the rest of the input and produce the rest of the output. This process happens reactively: output is only produced when an input arrives. The formal semantics of λ^{ST} is described in Section 3.2.

To represent stream transformers with multiple parallel and sequential inputs, we draw upon insights from proof theory. Both the types $s \cdot t$ and $s \| t$ are *product types*, in the sense that a stream of either of these types contains both the data of a stream of type s and a stream of type t—although the temporal structure differs between the two. A standard observation from proof theory is that, in situations where a logic or type theory includes two products with different structural properties, the corresponding typing judgment requires a context with two different *context formers*.[2]

The first context former, written with a comma ($\Gamma\,,\,\Delta$), describes inputs to a transformer arriving in parallel, one component structured according to Γ and the other according to Δ. The second context former, written with a semicolon ($\Gamma\,;\,\Delta$) describes inputs that will first arrive from the environment according to Γ, then according to Δ.

These interpretations are enforced by restricting the ways that these contexts can be manipulated using *structural rules*. Comma contexts can be manipulated in all the ways standard contexts can: their bindings can be reordered (from $\Gamma\,,\,\Delta$ to $\Delta\,,\,\Gamma$) duplicated, and dropped. Semicolon contexts, on the other hand, are ordered and affine: a context $\Gamma\,;\,\Delta$ cannot be freely rewritten to a context $\Delta\,;\,\Gamma$, and a context Γ cannot be duplicated into $\Gamma\,;\,\Gamma$. These restrictions enforce the interpretation of $\Gamma\,;\,\Delta$ as data arriving according to Γ and then Δ: to exchange them would be to allow a consumer to assume that the data is sent in the opposite order, and to duplicate would be to assume that the data input will be replayed.

Thus, part of our type system is *substructural*: the semicolon context former is ordered (no exchange) and affine (no contraction), while the comma context former is fully structural. Both context formers are associative, with the empty context serving as a unit for each. (The full list of structural rules can be found in the extended version [22].) Formally, stream contexts are drawn from the grammar at the top right of Figure 1.

3.1 Kernel Typing Rules

The typing rules for Kernel λ^{ST} are collected in Figure 1. The typing judgment, written $\Gamma \vdash e : s$, says that e is a stream transformer from a collection of streams structured like Γ to a single stream structured like s.

The most straightforward typing rule is the right rule for parallel (T-Par-R). It says that, from a context Γ, we can produce a stream of type $s \| t$ by producing s and t independently from Γ, using transformers e_1 and e_2. We write the combined transformer as a "parallel pair" (e_1, e_2). Semantically, it operates by copying the inputs arriving on Γ, passing the copies to e_1 and e_2, and merging the tagged outputs into a parallel stream. Similarly, the T-Cat-R rule is used to produce a stream of type $s \cdot t$. It uses a similar pairing syntax—if term e_1 has type s and e_2 has type t, then the "sequential pair" $(e_1; e_2)$ has type $s \cdot t$—but the context in the conclusion differs. Since e_1 needs to run before e_2, the part of the input stream that e_1 depends on must arrive before the part that e_2 depends on. Semantically, this term will operate by accepting data from the Γ part of the context and running e_1; once e_1 has produced its output it will switch to running e_2, consuming data from Δ.

[2]Such *bunched* contexts were first introduced in the Logic of Bunched Implication [63], the basis of modern separation logic [66]. Our bunched contexts differ from those of BI by the choice of structural rules: our substructural type former is affine ordered, while the BI one is linear.

$$s, t, r := 1 \mid \varepsilon \mid s \cdot t \mid s \| t \qquad \Gamma ::= \cdot \mid \Gamma, \Gamma \mid \Gamma; \Gamma \mid x : s$$

$$\frac{\Gamma \vdash e_1 : s \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash (e_1, e_2) : s \| t}\ \text{T-Par-R} \qquad \frac{\Gamma(x : s, y : t) \vdash e : r}{\Gamma(z : s \| t) \vdash \texttt{let}\ (x, y)\ =\ z\ \texttt{in}\ e : r}\ \text{T-Par-L}$$

$$\frac{\Gamma \vdash e_1 : s \qquad \Delta \vdash e_2 : t}{\Gamma\,; \Delta \vdash (e_1; e_2) : s \cdot t}\ \text{T-Cat-R} \qquad \frac{\Gamma(x : s\,; y : t) \vdash e : r}{\Gamma(z : s \cdot t) \vdash \texttt{let}_t\ (x; y)\ =\ z\ \texttt{in}\ e : r}\ \text{T-Cat-L}$$

$$\frac{}{\Gamma \vdash \texttt{sink} : \varepsilon}\ \text{T-Eps-R} \qquad \frac{}{\Gamma \vdash () : 1}\ \text{T-One-R} \qquad \frac{}{\Gamma(x : s) \vdash x : s}\ \text{T-Var}$$

$$\frac{\Gamma \leq \Gamma' \qquad \Gamma' \vdash e : s}{\Gamma \vdash e : s}\ \text{T-SubCtx}$$

Fig. 1. Kernel λ^{ST}syntax and typing rules

These right rules describe how to *produce* a stream of parallel or concatenation type. The corresponding left rules describe how to *use* a variable of one of these types appearing somewhere in the context. Syntactically, the terms take the form of let-bindings that deconstruct variables of type $s \cdot t$ (or $s \| t$) as pairs of variables of type s and t, connected by ; (or ,). We use the standard BI notation $\Gamma(-)$ for a context with a hole and write $\Gamma(\Delta)$ when this hole has been filled with the context Δ. In particular, $\Gamma(x : s)$ is a context with a distinguished variable x.

The T-Par-L rule says that if z is a variable of type $s \| t$ somewhere in the context, we can replace its binding with with a pair of bindings for variables x and y of types s and t and use these in a continuation term e of final type r. When typing e, the variables x and y appear in the same position as the original variable z, separated by a comma—i.e., x and y are assumed to arrive in parallel. Similarly, the rule T-Cat-L says that if a variable z of type $s \cdot t$ appears somewhere in the context, it can be let-bound to a pair of variables x and y of types s and t that are again used in the continuation e. This time, though, x and y are separated by a semicolon—i.e., the substream bound to x will arrive and be processed first, followed by the substream bound to y.

T-Eps-R and T-One-R are the right rules for the two base types, witnessed by the terms `sink` and `()`. Semantically, `sink` does nothing: it accepts inputs on Γ and produces no output. On the other hand, `()` emits a unit value as soon as it receives its first input and never emits anything else.

The variable rule (T-Var) says that if $x : s$ is a variable somewhere in the context, then we can simply send it along the output stream. Semantically, it works by dropping everything in the context except for the s-typed data for x, which it forwards along.

The rule T-SubCtx bundles together all of the structural rules as a subtyping relation on contexts. For example, the weakening rule for semicolon contexts is written $\Gamma\,; \Delta \leq \Gamma$ and the comma exchange rule is $\Gamma\,, \Delta \leq \Delta\,, \Gamma$.

Examples and Non-Examples. To show the typing rules in action, here are two small examples of transformers written in Kernel λ^{ST}, as well as three examples of programs that are *rejected* by the type system. The first example is a simple "parallel-swap" transformer, which accepts a stream z of type $s \| t$ and outputs a stream of type $t \| s$, swapping the parallel substreams:

$$z : s \| t \vdash \texttt{let}\ (x, y)\ =\ z\ \texttt{in}\ (y, x) : t \| s$$

It works by splitting the variable $z : s \| t$ into variables $x : s$ and $y : t$ and yielding a parallel pair with the order reversed.

$$\frac{}{\texttt{epsEmp} : \texttt{prefix}\,(\varepsilon)} \qquad \frac{}{\texttt{oneEmp} : \texttt{prefix}\,(1)} \qquad \frac{}{\texttt{oneFull} : \texttt{prefix}\,(1)}$$

$$\frac{p : \texttt{prefix}\,(s) \quad p' : \texttt{prefix}\,(t)}{\texttt{parPair}(p, p') : \texttt{prefix}\,(s \| t)} \qquad \frac{p : \texttt{prefix}\,(s)}{\texttt{catFst}(p) : \texttt{prefix}\,(s \cdot t)} \qquad \frac{p' : \texttt{prefix}\,(t) \quad p : \texttt{prefix}\,(s) \quad p\ \texttt{maximal}}{\texttt{catBoth}(p, p') : \texttt{prefix}\,(s \cdot t)}$$

Fig. 2. Prefixes for Types

The most important *non*-example is the lack of a corresponding "cat-swap" term, which would accept a stream z of type $s \cdot t$, and produce a stream of type $t \cdot s$. This program is undesirable because it is not implementable without a space leak. Implementing it requires the entire stream of type s to be saved in memory to emit it after the stream of type t.[3] The natural term for this program would be $\texttt{let}_s\ (x; y)\ =\ z\ \texttt{in}\ (y; x)$, but this does not typecheck. Applying the syntax-directed rules gets us to a point where we must show that y has type t in a context with only x and that x has type s in a context with only y. This is because the T-Cat-R splits the context, but the variables are listed in the opposite order from what we'd need. The lack of a structural rule to let us permute the x and y in the context means that there is nothing we can do here, and a typechecker will reject this program.

$$\dfrac{\dfrac{\dfrac{\lightning}{x : s \vdash y : t}\ \text{T-Var} \quad \dfrac{\lightning}{y : t \vdash x : s}\ \text{T-Var}}{x : s\,; y : t \vdash (y; x) : t \cdot s}\ \text{T-Cat-R}}{z : s \cdot t \vdash \texttt{let}_s\ (x; y)\ =\ z\ \texttt{in}\ (y; x) : t \cdot s}\ \text{T-Cat-L}$$

The second example is a "broadcast" transformer, which takes a variable $x : s$ and outputs a stream of type $s \| s$, duplicating the variable and sending it out to two parallel outputs: $x : s \vdash (x, x) : s \| s$.

The second non-example is the "replay" transformer, which would (if it existed) take a variable $x : s$ and produce a stream $s \cdot s$ that repeats the input stream twice. This is the concat-equivalent of the broadcast transformer, and it is undesirable for the same reason as the cat-swap program: it would require saving the entire incoming stream of type s in order to replay it. This time, the failure of the natural term $(x; x)$ to typecheck comes down to the lack of a contraction rule for semicolon contexts: we are not permitted to turn a context $x : s$ into a context $x : s\,; x : s$.

The last non-example is a "tie-breaking" transformer, which would take a stream $z : \texttt{Int} \| \texttt{Int}$ of two ints in parallel and produce a stream of type `Int` by forwarding along the `Int` that arrives first. This program, like others that require inspecting the interleaving of substreams in a stream of type $s \| t$, is not expressible. In Section 3.3, we'll prove that a well-typed program cannot implement this behavior.

3.2 Prefixes and Semantics

We next define the semantics of Kernel λ^{ST}. The natural notion of "values" in this semantics is finite prefixes of streams, and the meaning of a well-typed term $\Gamma \vdash e : s$ is a function that accepts an environment mapping variables in Γ to prefixes of streams and produces a prefix of a stream of type s. Because the streams that λ^{ST} programs operate over are more structured than traditional homogeneous streams—including cross-over punctuation in streams of type $s \cdot t$ and disambiguating tags in streams of type $s \| t$—the prefixes are also more structured. That is, a prefix in λ^{ST} is not a simple sequence of data items, but a structured value whose possible shapes are determined by its type [58].

[3] A program with this behavior is actually implementable in λ^{ST}, but only with a special additional construct—see Section 4.5—ensuring that leaky programs like this one cannot be written accidentally.

$$\frac{}{\eta \;:\; \mathsf{env}\,(\cdot)} \qquad \frac{\eta(x) \mapsto p \qquad p \;:\; \mathtt{prefix}\,(s)}{\eta \;:\; \mathsf{env}\,(x : s)} \qquad \frac{\eta \;:\; \mathsf{env}\,(\Gamma) \qquad \eta \;:\; \mathsf{env}\,(\Delta)}{\eta \;:\; \mathsf{env}\,(\Gamma\,, \Delta)}$$

$$\frac{\eta \;:\; \mathsf{env}\,(\Gamma) \qquad \eta \;:\; \mathsf{env}\,(\Delta) \qquad (\eta\;\mathtt{maximalOn}\;\Gamma) \vee (\eta\;\mathtt{emptyOn}\;\Delta)}{\eta \;:\; \mathsf{env}\,(\Gamma\,; \Delta)}$$

Fig. 3. Environments for Contexts

There are two prefixes of a stream of type 1: the empty prefix, written `oneEmp`, and the prefix containing the single element (), written `oneFull`. Similarly, the unique stream of type ε has a single prefix, the empty prefix, which we write `epsEmp`.

What about $s \| t$? A parallel stream of type $s \| t$ is conceptually a pair of independent streams of type s and t, so a prefix of a parallel stream should be a pair $\mathtt{parPair}(p_1, p_2)$, where p_1 is a prefix of a stream of type s, and p_2 is a prefix of a stream of type t. Crucially, this definition encodes no information about any interleaving of p_1 and p_2: the prefix $\mathtt{parPair}(p_1, p_2)$ equally represents a situation where all of p_1 arrived first and then all of p_2, one where p_2 arrived before p_1, and many others where the elements of p_1 and p_2 arrived in some interleaved order. In a nutshell, this definition is what guarantees deterministic processing. By representing all possible interleavings using the same prefix value, we ensure that a transformer that operates on these values cannot possibly depend on ordering information that isn't present in the type.

Finally, let's consider the prefixes of streams of type $s \cdot t$. One case is a prefix that only includes data from s because it cuts off before reaching the point where the $s \cdot t$ stream stops carrying elements of s and starts on t. We write such a prefix as $\mathtt{catFst}(p)$, with p a prefix of type s. The other case is where the prefix does include the crossover point—i.e., it consists of a "completed" prefix of s plus a prefix of t. We write this as $\mathtt{catBoth}(p, p')$, with p a prefix of s and p' a prefix of t. The requirement that p be completed is formalized by the judgment p `maximal`, which ensures that the prefix p describes an entire completed stream (see the extended version [22] for details). We formalize all these possibilities as a judgment $p \;:\; \mathtt{prefix}\;(s)$, shown in Figure 2.

Every type s has a distinguished *empty prefix*, written $\mathtt{emp}_s$ and defined by straightforward recursion on s (see the extended version [22]). We then lift the idea of prefixes from types to contexts, defining an *environment* η for a context Γ to be a mapping from the variables $x : s$ in Γ to prefixes of the corresponding types s; we write this with a judgment $\eta \;:\; \mathsf{env}\,(\Gamma)$ (Figure 3). Besides ensuring that η has well-typed bindings for all variables, the judgment ensures that the prefixes respect the order structure of the context. In particular, an environment η for a semicolon context $\Gamma\,; \Delta$ must assign prefixes *in order*: the prefixes for Γ, the earlier part of the context, must all be complete before any of the prefixes for Δ can begin. In other words, either η assigns maximal prefixes to every variable in Γ—which we write $\eta\;\mathtt{maximalOn}\;\Gamma$—or η assigns empty prefixes to every variable in Δ—which we write $\eta\;\mathtt{emptyOn}\;\Delta$.

One might worry that these structured stream prefixes might be incompatible with a future distributed implementation atop an existing stream processing substrate. Fortunately, they are not: by viewing a λ^{ST} stream as a series of single-event prefixes, each consisting of a data item plus some extra tag bits, we can recover the traditional homogeneous view. Moreover, this wire representation incurs only a constant overhead: the maximum size of the tag bits on a stream element of type s is bounded by the syntactic depth of s (see the extended version [22] for details).

Semantics. We next describe how well-typed λ^{ST} terms behave with an operational semantics. Given a well-typed term $\Gamma \vdash e : s$ and an input environment $\eta \;:\; \mathsf{env}\,(\Gamma)$, this semantics describes how to run e with η to produce an output prefix p. It also describes how to produce a "resultant" term

e' that is ready to continue the computation once further data arrives on the input stream data. Formally, the semantics is given by a judgment $\eta \Rightarrow e \downarrow e' \Rightarrow p$, pronounced "running the core term e on the input environment η yields the output prefix p and steps to e'." The rules for this judgment are gathered in Figure 4 and described below; the full set of rules for all of λ^{ST} can be found in [22].

The following theorem establishes the soundness of the Kernel λ^{ST} semantics, formalizing the intuitive description given above: If we run a well-typed core term e on an environment η of the context type, it will return a prefix p with the result type s and step to a term e' that is well typed in context "the rest of Γ" after η and has type "the rest of s" after p. The "rest" of a type (or context) after a prefix (or environment) is, intuitively, its *derivative* with respect to the prefix (or environment), in the sense of standard Brzozowski derivatives of regular expressions [16]—we make this formal in Section 3.2. Most critically, the types of the variables in e and e' are different: if x has type s in e, then x has type $\delta_{\eta(x)} s$ in e', having already consumed $\eta(x)$.

Theorem 3.1 (Soundness of the Kernel λ^{ST} Semantics). *Suppose:* $\Gamma \vdash e : s$ *and* $\eta : \mathsf{env}(\Gamma)$. *Then there are* p *and* e' *such that* $\eta \Rightarrow e \downarrow e' \Rightarrow p$, *with* $p : \mathtt{prefix}(s)$ *and* $\delta_\eta(\Gamma) \vdash e' : \delta_p(s)$

See the appendix of the extended paper [22] for the proof of soundness for Full λ^{ST}.

In light of the soundness theorem, the operational semantics can be thought of as defining a reactive state machine. Well-typed terms $\Gamma \vdash e : s$ are the states, while the semantic judgment defines the transition function: when new inputs η arrive, the machine produces an output prefix p and steps to a new state $\delta_\eta(\Gamma) \vdash e' : \delta_p(s)$. This form of semantics—a state machine with terms themselves as states, typed by derivatives—was pioneered by the Esterel programming language [14].

Semantics of the Right Rules. The right rules for parallel and concatenation are the simplest to understand. For S-Par-R, we accept an environment η and use it to run the component terms e_1 and e_2, independently producing outputs p_1 and p_2 and stepping to new terms e_1' and e_2'. The pair term (e_1, e_2) then steps to (e_1', e_2') and produces the output $\mathtt{parPair}(p_1, p_2)$.

There are two rules, S-Cat-R-1 and S-Cat-R-2, for running the concatenation pair $(e_1; e_2) : s \cdot t$. In either case, we begin by running e_1 with the environment η, producing a prefix p and term e_1'. If p is not maximal, we stop there: more of the input is needed for the first component to produce the rest of s, so it is not yet time to start running e_2 to produce t. This case is handled by S-Cat-R-1, where the resulting term is $(e_1'; e_2)$ and the output prefix is $\mathtt{catFst}(p)$. On the other hand, if p is maximal, then we also run e_2, which steps to e_2' and produces a prefix p' using rule S-Cat-R-2; the entire term then outputs $\mathtt{catBoth}(p, p')$ and steps to e_2'. Note that the pair is eliminated in the process: we step from $(e_1; e_2)$ to just e_2'. This is because we are done producing the s part of the $s \cdot t$, and so a subsequent step of evaluation only has to run e_2' to produce the rest of the t.

Semantics of Variables. The variable semantics S-Var is a simple lookup. We find the prefix bound to the variable x in the environment, return it, and then step to x itself.

Semantics of Left Rules. The left rules for concatenation and parallel are similar, both accepting an environment η with a binding for $z : s \otimes t$ where $\otimes$ is one of the two products, binding variables x and y of types s and t to the two components of the product, and using the updated environment to run the continuation term. In the case of the left rule for parallel (S-Par-L), looking up z of type $s \| t$ will always yield a prefix $\mathtt{parPair}(p_1, p_2)$. The rule binds p_1 to x and p_2 to y and runs the continuation term, stepping to e' and producing the output prefix p. Then the whole term steps to $\mathtt{let}\ (x, y) = z\ \mathtt{in}\ e'$ and produces p.

$$\frac{\eta(x) \mapsto p}{\eta \Rightarrow x \downarrow x \Rightarrow p}\ \text{S-Var} \qquad \frac{\eta \Rightarrow e_1 \downarrow e_1' \Rightarrow p_1 \qquad \eta \Rightarrow e_2 \downarrow e_2' \Rightarrow p_2}{\eta \Rightarrow (e_1, e_2) \downarrow (e_1', e_2') \Rightarrow \mathtt{parPair}(p_1, p_2)}\ \text{S-Par-R}$$

$$\frac{\eta(z) \mapsto \mathtt{parPair}(p_1, p_2) \qquad \eta[x \mapsto p_1, y \mapsto p_2] \Rightarrow e \downarrow e' \Rightarrow p'}{\eta \Rightarrow \mathtt{let}\ (x, y)\ =\ z\ \mathtt{in}\ e \downarrow \mathtt{let}\ (x, y)\ =\ z\ \mathtt{in}\ e' \Rightarrow p'}\ \text{S-Par-L}$$

$$\frac{\eta \Rightarrow e_1 \downarrow e_1' \Rightarrow p \qquad \neg\,(p\ \mathtt{maximal})}{\eta \Rightarrow (e_1; e_2) \downarrow (e_1'; e_2) \Rightarrow \mathtt{catFst}(p)}\ \text{S-Cat-R-1}$$

$$\frac{\eta \Rightarrow e_1 \downarrow e_1' \Rightarrow p \qquad p\ \mathtt{maximal} \qquad \eta \Rightarrow e_2 \downarrow e_2' \Rightarrow p'}{\eta \Rightarrow (e_1; e_2) \downarrow e_2' \Rightarrow \mathtt{catBoth}(p, p')}\ \text{S-Cat-R-2}$$

$$\frac{\eta(z) \mapsto \mathtt{catFst}(p) \qquad \eta[x \mapsto p, y \mapsto \mathtt{emp}_t] \Rightarrow e \downarrow e' \Rightarrow p'}{\eta \Rightarrow \mathtt{let}_t\ (x; y)\ =\ z\ \mathtt{in}\ e \downarrow \mathtt{let}_t\ (x; y)\ =\ z\ \mathtt{in}\ e' \Rightarrow p'}\ \text{S-Cat-L-1}$$

$$\frac{\eta(z) \mapsto \mathtt{catBoth}(p, p') \qquad \eta[x \mapsto p, y \mapsto p'] \Rightarrow e \downarrow e' \Rightarrow p''}{\eta \Rightarrow \mathtt{let}_t\ (x; y)\ =\ z\ \mathtt{in}\ e \downarrow \mathtt{let}\ x = \mathtt{sink}_p\ \mathtt{in}\ e'[z/y] \Rightarrow p''}\ \text{S-Cat-L-2}$$

$$\frac{}{\eta \Rightarrow \mathtt{sink} \downarrow \mathtt{sink} \Rightarrow \mathtt{epsEmp}}\ \text{S-Eps-R}$$

$$\frac{}{\eta \Rightarrow () \downarrow \mathtt{sink} \Rightarrow \mathtt{oneFull}}\ \text{S-One-R}$$

Fig. 4. Incremental semantics of Kernel λ^{ST}

The left rule for concatenation has two cases, depending on what kind of prefix comes back from the lookup for z. If the lookup yields is $\mathtt{catFst}(p)$, then the rule S-Cat-L-1 applies. Since no data for y has arrived, we bind y to $\mathtt{emp}_t$, the empty prefix of type t, and run the continuation.[4] If the result comes back as $\mathtt{catBoth}(p, p')$, then the rule S-Cat-L-2 applies, so we run the continuation with x and y bound to p and p'.

Both rules output the prefix obtained from running the continuation, but they step to different resulting terms. If $\eta(z) = \mathtt{catFst}(p)$, then the resulting term must be another use of Cat-L: the variable z still expects to get some more of the first component of the concatenation, and then the second component. If $\eta(z) = \mathtt{catBoth}(p, p')$ on the other hand, the z stream has crossed over to the second part. In this case, we close over the (now not-needed) x variable in e' and connect z to the y input of e' by substituting y for z.

Derivatives. When $p : \mathtt{prefix}(s)$, we write $\delta_p(s)$ for the derivative [16] of s by p—the type of streams that result after a prefix of type p has been "chopped off" the beginning of a stream of type s. Because this operation is partial—$\delta_p(s)$ is only defined when $p : \mathtt{prefix}(s)$—we formally define this as a a 3-place relation, written as $\delta_p(s) \sim s'$ and pronounced as "the derivative of s with respect to p is s'" (see Figure 5).

The derivative of the type 1 with respect to the empty prefix `oneEmp` is 1 (the rest of the stream is the entire stream), and its derivative with respect to the full prefix `oneFull` is ε (there is no more

[4]This need to compute $\mathtt{emp}_t$ from t at runtime to bind to $y \mapsto \mathtt{emp}_t$ is the reason that the term for `T-Cat-L`, $\mathtt{let}_t\ (x; y)\ =\ z\ \mathtt{in}\ e$, includes a t in the syntax. In Section 4, the case analysis expressions for star types and sum types will have similar annotations for the same reason.

$$\frac{}{\delta_{\texttt{epsEmp}}\,(\varepsilon) \sim \varepsilon} \qquad \frac{}{\delta_{\texttt{oneEmp}}\,(1) \sim 1} \qquad \frac{}{\delta_{\texttt{oneFull}}\,(1) \sim \varepsilon} \qquad \frac{\delta_p\,(s) \sim s'}{\delta_{\texttt{catFst}(p)}\,(s \cdot t) \sim s' \cdot t}$$

$$\frac{\delta_{p'}\,(t) \sim t'}{\delta_{\texttt{catBoth}(p,'p)}\,(s \cdot t) \sim t'} \qquad \frac{\delta_p\,(s) \sim s' \qquad \delta_{p'}\,(t) \sim t'}{\delta_{\texttt{parPair}(p,p')}\,(s \| t) \sim s' \| t'}$$

Fig. 5. Derivatives

stream left after the unit element has arrived). For parallel, the derivative is taken component-wise. The interesting cases are those for the concatenation type. If the prefix has the form $\texttt{catFst}(p)$, the derivative $\delta_{\texttt{catFst}(p)}\,(s \cdot t)$ is $\left(\delta_p\,(s)\right) \cdot t$, i.e., some of the s has gone by but not all, and once it does we still expect t to come after it. On the other hand, if the prefix has the form $\texttt{catBoth}(p, p')$, the derivative $\delta_{\texttt{catBoth}(p,p')}\,(s \cdot t)$ is just $\delta_{p'}\,(t)$, i.e., the s component is complete, and the rest of the stream is just the part of t after p'.

This definition is lifted to contexts and environments pointwise: if $x : s$ is a variable in Γ, the derivative of $\delta_\eta\,(\Gamma)$ has $x : \delta_{\eta(x)}\,(s)$ in the same location.

3.3 The Homomorphism Property and Determinism

The semantics is designed to run a stream transformer on "input chunks" of any size, from individual input events one at a time all the way up to the entire stream at once. The cost of this flexibility is that it raises the question of *coherence*—i.e., whether we are guaranteed to arrive at the *same* final output depending on how we carve up a transformer's input into a series of prefixes. Fortunately, this is indeed guaranteed. Coherence is a corollary of our main technical result: a *homomorphism theorem* that says running a term e on an environment η and then running the resulting term e' on an environment η' of appropriate type produces the same end result as running e on the combined environment (cf. [58]).

Theorem 3.2 (Homomorphism Theorem). *Suppose (1)* $\Gamma \vdash e : s$*, (2)* $\eta : \texttt{env}\,(\Gamma)$*, (3)* $\eta' : \texttt{env}\left(\delta_\eta\,(\Gamma)\right)$*, (4)* $p : \texttt{prefix}\,(s)$*, (5)* $p' : \texttt{prefix}\left(\delta_p\,(s)\right)$*, (6)* $\eta \Rightarrow e \downarrow e' \Rightarrow p$*, and (7)* $\eta' \Rightarrow e' \downarrow e'' \Rightarrow p'$*. Then, if* $\eta \cdot \eta' \Rightarrow e \downarrow e''' \Rightarrow p''$*, we have* $p'' = p \cdot p'$*, and* $e''' = e'$

The operation $p \cdot p'$ here is *prefix concatenation*, which takes a prefix p of type s and a prefix p' of type $\delta_p\,(s)$ and produces the prefix of type s that is first p and then p'. Formally, this is defined as a 4-place partial inductive relation $p \cdot p' \sim p''$, which is defined when p and p' have types s and $\delta_p\,(s)$, respectively. The operation $\eta \cdot \eta' \sim \eta''$ does the same for environments. .

The homomorphism theorem not only justifies running the semantics on prefixes of any size; it also implies deterministic processing of parallel streams. Intuitively, determinism states that the results of a stream transformer do not depend on the particular order in which parallel data arrives. We formalize this through the following scenario. Suppose $\Gamma, \Gamma' \vdash e : s$ is a term with two parallel contexts serving as its input, and suppose that η is an environment for Γ, Γ'. Write $\eta_1 = \eta|_\Gamma$ and $\eta_2 = \eta|_{\Gamma'}$, for the restrictions of η to the variables in Γ and Γ', respectively. There are two different ways of running e on this data. One is to first run e on $\eta_1 \cup \texttt{emp}_{\Gamma'}$ (which has η_1 bindings for Γ and then the empty prefix for everything in Γ') and then run the resulting term on $\eta_2 \cup \texttt{emp}_\Gamma$ (with an empty prefixes for Γ). The other does the opposite, first running e on $\eta_2 \cup \texttt{emp}_\Gamma$ and then running the resulting term on $\eta_1 \cup \texttt{emp}_{\Gamma'}$. Determinism says that these strategies produce equal results.

Theorem 3.3 (Determinism). *Suppose (1)* $\Gamma, \Gamma' \vdash e : s$, *(2)* η : $\mathtt{prefix}\,(\Gamma, \Gamma')$, *(3)* $\eta|_\Gamma \cup \mathtt{emp}_{\Gamma'} \Rightarrow e \downarrow e_1 \Rightarrow p_1$ *and* $\eta|_{\Gamma'} \cup \mathtt{emp}_\Gamma \Rightarrow e_1 \downarrow e_2 \Rightarrow p_2$, *(4)* $\eta|_{\Gamma'} \cup \mathtt{emp}_\Gamma \Rightarrow e \downarrow e_1' \Rightarrow p_1'$ *and* $\eta|_\Gamma \cup \mathtt{emp}_{\Gamma'} \Rightarrow e_1' \downarrow e_2' \Rightarrow p_2'$. *Then* $e_2 = e_2'$ *and* $p_1 \cdot p_2 = p_1' \cdot p_2'$.

See the appendix of the extended version [22] for the proof. To intuitively see how this theorem follows from homomorphism, note that prefixes are canonical representatives of equivalence classes of sequences of stream elements, up to the possible reorderings defined by their type [70]. The homomorphism theorem then guarantees that these normal forms are processed compositionally, and so are independent of the actual temporal ordering of parallel data—it suffices to compute on the combined normal forms from the two steps.

4 FULL λ^{ST}

We now sketch the remaining types and terms of λ^{ST} that are not part of Kernel λ^{ST}.

4.1 Sums

Sum types in λ^{ST}, written $s + t$, are *tagged* unions: a stream of type $s + t$ is either a stream of type s or a stream of type t, and a consumer can tell which. Streams of type s are not the same as streams of type $s + s$, and streams of type $s + t$ are isomorphic to, but not identical to, streams of type $t + s$. Operationally, a producer of a sum stream sends a tag bit before sending the rest of the stream, to tell downstream consumers which side to expect. Conversely, a consumer of $s + t$ first reads the bit to learn which it is getting next.

A prefix of $s + t$ can be a prefix of one of s or one of t, written $\mathtt{sumInl}(p)$ or $\mathtt{sumInr}(p)$, or it can be $\mathtt{sumEmp}$, the empty prefix of type $s + t$, which does not even include the initial tag bit. The derivatives with respect to these prefixes are defined as follows: (a) the empty prefix takes nothing off the type ($\delta_{\mathtt{sumEmp}}\,(s + t) = s + t$) and (b) the two injections reduce to taking the derivative of the corresponding branch of the sum ($\delta_{\mathtt{sumInl}(p)}\,(s + t) = \delta_p\,(s)$ and $\delta_{\mathtt{sumInr}(p)}\,(s + t) = \delta_p\,(t)$).

The typing rules for sums are the normal injections

$$\frac{\Gamma \vdash e : s}{\Gamma \vdash \mathtt{inl}\,(e) : s + t}\ \text{T-Sum-R-1} \qquad \frac{\Gamma(x : s) \vdash e_1 : r \qquad \Gamma(y : t) \vdash e_2 : r}{\Gamma(z : s + t) \vdash \mathtt{case}_r(z, x.e_1, y.e_2) : r}\ \text{T-Sum-L-Surf}$$

on the right (T-Sum-R-1 and a symmetric rule T-Sum-R-2) and a case analysis rule on the left (T-Sum-L-Surf). The right rules operate by prepending their respective tags and then running the embedded terms. The left rule does case analysis: if the incoming stream z comes from the left of the sum, it is processed with e_1; if from the right, e_2. To run a sum case term, the semantics must dispatch on the tag that says if the stream z being destructed is a left or a right. But the prefix z might not include a tag, if only data from the surrounding context has arrived. In this case, z will map to $\mathtt{sumEmp}$, and we have no way of determining which branch to run. The solution is to run neither! Instead, we hold on to the environment, saving *all* incoming data to the program until the tag arrives. Once we get a prefix that includes the tag, we continue by running the corresponding branch with the accumulated inputs. Note that this buffering is necessarily a blocking operation. [5]

All this requires a slightly generalized typing rule (T-Sum-L) that includes a *buffer environment* η : $\mathtt{env}\,(\Gamma(z : s + t))$ of the context type in the term. This buffer holds all of the input data we've seen so far. As prefixes arrive,

$$\frac{\eta : \mathtt{env}\,(\Gamma(z : s + t)) \qquad \Gamma(x : s) \vdash e_1 : r \qquad \Gamma(y : t) \vdash e_2 : r}{\delta_\eta\,(\Gamma(z : s + t)) \vdash \mathtt{case}_r\,(\eta; z, x.e_1, y.e_2) : r}\ \text{T-Sum-L}$$

[5] Depending on the rest of the context, it could also require unbounded memory! Fortunately, we believe we can detect this, and flag it as a warning to the user: running a case on $z : s + t$ in a context $\Gamma(z : s + t)$ could require buffering all variables to the left of z or in parallel with z in the context. Unbounded memory is required if and only if any of those variables have star type. We hope to demonstrate this formally in future work.

we append to this buffer until we get the tag. Accordingly, the context in this rule is $\delta_\eta\ (\Gamma(z : s + t))$: the term is typed in the context consisting of everything after the part of the stream that has so far been buffered.

Fortunately, the only typing rule that a λ^{ST} programmer needs to concern themselves with is T-Sum-L-Surf. While writing the program, and before it runs, the buffer is empty ($\eta = \mathtt{emp}_{\Gamma(z:s+t)}$). In this case, the $\delta_\eta\ (\Gamma(z : s + t)) = \Gamma(z : s + t)$, and so the generalized rule T-Sum-L simplifies to the "surface" rule, T-Sum-L-Surf. Full details can be found in the extended version [22].

4.2 Star

Full λ^{ST} also includes a type constructor for unbounded streams, written $s^\star$ because it is inspired by the Kleene star from the theory of regular languages. (We do not need to distinguish between unbounded finite streams and "truly infinite" ones, because our operational semantics is based on prefixes: we're always only operating on "the first part" of the input stream, and it doesn't matter whether the part we haven't seen yet is finite or infinite.) The type $s^\star$ describes a stream that consists of zero or more sub-streams of type s, in sequence.

In ordinary regular languages, $r^\star$ is equal to $\varepsilon + r \cdot r^\star$. In the language of stream types, this equation says that a stream of type $s^\star$ is either empty (ε) or a stream of type s followed by another stream of type $s^\star$—i.e., $s^\star$ can be understood as the least fixpoint of the stream type operator $x \mapsto \varepsilon + s \cdot x$. The definitions of prefixes and typing rules for star all follow from this perspective.

In particular, $\mathtt{prefix}(s^\star) = \mathtt{prefix}(\varepsilon + s \cdot s^\star)$. The empty prefix of type $s^\star$, written `starEmp`, is effectively the empty prefix of the sum that makes up $s^\star$. The second form of prefix—the "done" prefix of type $s^\star$—is written `starDone`. It corresponds to the left injection of the sum, and receiving it means that the stream has ended. Note that, despite containing no s data, this prefix is not *empty*: it conveys the information that the stream is complete. The final two cases correspond to the right injection of the sum, i.e., a prefix of type $s \cdot s^\star$. This is either $\mathtt{starFirst}(p)$, with p a prefix of s, or $\mathtt{starRest}(p, p')$, with p a maximal prefix of type s and p' another prefix of $s^\star$.

For derivatives, the empty prefix leaves the type as-is ($\delta_{\mathtt{starEmp}}\ (s^\star) = s^\star$). Because no data will arrive after the done prefix, the derivative of $s^\star$ with respect to `starDone` is ε. In the case for $\mathtt{starFirst}(p)$, after some of an s has been received, the remainder of $s^\star$ looks like the remainder of the first s followed by some more $s^\star$, so the derivative is defined as $\delta_{\mathtt{starFirst}(p)}\ (s^\star) = (\delta_p\ (s)) \cdot s^\star$. Finally, $\delta_{\mathtt{starRest}(p,p')}\ (s^\star) = \delta_{p'}\ (s^\star)$.

The typing rules for star are again motivated by the analogy with lists. There are right rules for `nil` and `cons` and a case analysis principle for the left rule. The "nil" rule T-Star-R-1 corresponds to the left injection into the sum $s^\star = \varepsilon + s \cdot s^\star$: from any context, we can produce $s^\star$ by simply ending the stream. The "cons" rule T-Star-R-2 is the right injection: from a context $\Gamma;\Delta$, we can produce an $s^\star$ by producing one s from Γ and the remaining $s^\star$ from Δ. Operationally, this should run the same way as the T-Cat-R rule: by first running e_1, and if an entire s is produced, continuing by running e_2 to produce some prefix of the tail. The T-Star-L rule is a case analysis principle for streams of star type: either such a stream is empty, or else it comprises one s followed by an $s^\star$. The fact that the head s will come first and the tail $s^\star$ later tells us that the variables $x : s$ and $xs : s^\star$ should be separated by a semicolon in the context. Like T-Sum-L, this rule includes a buffer, collecting input environments until the prefix bound to z is enough to make the decision for which branch of the case to run.

$$\frac{}{\Gamma \vdash \mathtt{nil} : s^\star}\ \text{T-Star-R-1} \qquad \frac{\Gamma \vdash e_1 : s \qquad \Delta \vdash e_2 : s^\star}{\Gamma;\Delta \vdash e_1 :: e_2 : s^\star}\ \text{T-Star-R-2}$$

The semantics of the right rules are straightforward: the rules for T-Star-R-1 are like those for T-Eps-R, while the rules for T-Star-R-2 are like those for T-Cat-R. The semantics of T-Star-L is

just like T-Sum-L, buffering input prefixes until either (a) we get $z \mapsto \texttt{starDone}$, at which point we run e_1, or (b) we get $z \mapsto \texttt{starFirst}(p)$ or $z \mapsto \texttt{starRest}(p, p')$, in which case we run e_2. For full details, see the extended version [22].

4.3 Let-Binding

Full λ^{ST} also allows for more general let-binding. Given a transformer e whose output is used in the input of another term e', we can compose them to form a single term $\texttt{let}\, x = e\, \texttt{in}\, e'$ that operates as the sequential composition of e followed by e'. The rules for this construct are in Figure 6. Note that this sequencing is not the same kind of sequencing as in a concat-pair $(e; e')$. The latter produces data that follows the sequential pattern $s \cdot t$, while the former is sequential composition of code. When a let binding is run, both terms are evaluated, and the output of the first is passed to the input of the second. An important point to note is that this semantics is non-blocking: even if e produces the empty prefix, we still run e', potentially producing output.

$$\frac{\Gamma(\cdot) \vdash e_1 : r \qquad \Gamma(x : s; xs : s^\star) \vdash e_2 : r \qquad \eta \;:\; \mathsf{env}\left(\Gamma(z : s^\star)\right)}{\delta_\eta\left(\Gamma(z : s^\star)\right) \vdash \mathsf{case}_{s,r}\,(p; z, e_1, x.xs.e_2) : r}\ \text{T-Star-L}$$

The semantic rule S-Let for let-binding (in Figure 6) is a straightforward encoding of this behavior. Given the input environment η, we run the term e, bind the resulting prefix p to x, and run the continuation e', returning its output. The resultant term is another let-binding between the resultant terms of e and e'.

$$\frac{\Delta \vdash e : s \qquad \Gamma(x : s) \vdash e' : t \qquad e\ \texttt{inert}}{\Gamma(\Delta) \vdash \texttt{let}\, x = e\, \texttt{in}\, e' : t}\ \text{T-Let}$$

$$\frac{\eta \Rightarrow e_1 \downarrow e_1' \Rightarrow p \qquad \eta[x \mapsto p] \Rightarrow e_2 \downarrow e_2' \Rightarrow p'}{\eta \Rightarrow \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \downarrow \texttt{let}\, x = e_1'\, \texttt{in}\, e_2' \Rightarrow p'}\ \text{S-Let}$$

Fig. 6. Rules for Let-Bindings

The typing rule T-Let says that if e has type s in context Δ and e' has type t in a context $\Gamma(x : s)$ with a variable of type s, we can form the let-binding term $\texttt{let}\, x = e\, \texttt{in}\, e'$, which has type t in context $\Gamma(\Delta)$. The soundness of the semantics rule S-Let depends on a subtle requirement: e must not produce nonempty output until e' is ready to accept it. This is enforced by the third premise of the T-Let rule, which states that e must be *inert*: it only produces nonempty output when given nonempty input. This restriction rules out let-bindings such as $\texttt{let}\, x = ()\, \texttt{in}\, e'$, since the semantics of $()$ always produces nonempty output (namely $\texttt{oneFull}$), even when given an environment mapping every variable to an empty prefix[6]. In actuality, inertness is not a purely syntactic condition on terms, but depends also on typing information. To this end, inertness is tracked like an effect through the type system: see the appendix [22] for details.

4.4 Recursion

To write interesting transformers over $s^\star$ streams, we provide a way to define transformers recursively. Adding a traditional general recursion operator $\texttt{fix}(x.e)$ does not work in our context, as arrow types are required to define functions this way. We instead add explicit term-level recursion and recursive call operators. The program $\texttt{fix}\left(e_{\texttt{args}}\right).(e)$ defines a recursive transformer with body e and initial arguments $e_{\texttt{args}}$. Recursive calls are made inside the body e with a term $\texttt{rec}\left(e_{\texttt{args}}\right)$, which calls the function being defined with arguments $e_{\texttt{args}}$. This back-reference works in the same way that uses of the variable x in the body of a traditional fix point $\texttt{fix}(x.e)$ refer to the term $\texttt{fix}(x.e)$ itself. This function-free approach is approach is inspired by the concept of *cyclic proofs* [15, 24, 29] from proof theory, where derivations may refer back to themselves. Alternatively,

[6]Because such let-bindings are essentially trivial, we expect that they can be eliminated — see Section 7 for more discussion.

one can think of this construction as defining our terms and proof trees as infinite *coinductive* trees; then the term-level `fix` operator defines terms as *co*fixpoints.

In brief, to typecheck a fixpoint term, we simply type its body e, assuming that all instances of the `rec` in e have the same type as the fixpoint itself. Then, to run a fixpoint term $\texttt{fix}\,(e_{\texttt{args}})\,.\,(e)$, the rule unfolds the recursion one step by substituting the body e for instances of `rec` in itself, then runs the resulting term, binding all of the arguments to their variables. Full details of the typing rules and semantics of fixpoints can be found in the extended version [22].

Naturally, this semantics can lead to non-termination, as $\texttt{fix(rec)}$ unfolds to itself.[7] To bound the depth of evaluation, we *step index* both semantic judgments by adding a fuel parameter that decreases when we unfold a `fix`. The semantic judgment then looks like $\eta \Rightarrow e \downarrow^n e' \Rightarrow p$: when we run e on η, it steps to e' producing p and unfolding at most n uses of `fix` along the way.

4.5 Stateful Transformers

In the λ^{ST} typing judgment $\Gamma \vdash e : s$, the variables in Γ range over *future values* that have yet to arrive at the transformer e. The ordered nature of semicolon contexts means that variables further to the right in Γ correspond to data that will arrive further in the future. This imposes a strong restriction on programming: if earlier values in the stream are used at all, they must be used *before* later values; once a value in the stream has "gone by," there is no way to refer to it again. By using variables from the Γ context, a term e can refer to values that will arrive in the future; but it has no way of referring to values that have arrived in the *past*. This limitation is by design: from a programming perspective, referring to variables from the past requires *memory*, which is a resource to be carefully managed in streaming contexts. Of course, while some important streaming functions (e.g., map and filter) can get by without state, but many others (e.g., "running sums") require it. In this section, we add support for stateful stream transformers.

To maintain state from the past, we extend the typing judgment of λ^{ST} to include a second context, Ω, called the *historical context*, which gives types to variables bound to values stored in memory. We write $\Omega \mid \Gamma \vdash e : s$ to mean "e has type s in context Γ and historical context Ω".

What types do variables in the historical context have? Once a complete stream of type $(\texttt{Int}^\star \| \texttt{Int}^\star) \cdot \texttt{Int}^\star$ has been received and is stored in memory, we may as well regard the data as a value of the standard type $(\texttt{list(Int)} \times \texttt{list(Int)}) \times \texttt{list(Int)}$ from the simply typed lambda-calculus (STLC). In other words, parts of streams that *will* arrive in the future have stream types, parts of streams that *have* arrived in the past can be given standard STLC types. The "flattening" operation $\langle s \rangle$ transforms stream types into STLC types. The interesting cases of its definition are $\langle s \cdot t \rangle = \langle s \| t \rangle = \langle s \rangle \times \langle t \rangle$ and $\langle s^\star \rangle = \texttt{list}\,(\langle s \rangle)$.

The historical context is a fully structural: $\Omega ::= \cdot \mid \Omega, x : A$, where the types A are drawn from some set of conventional lambda-calculus types including at least products, sums, a unit, and a list type. Operationally, the historical context behaves like a standard context in a functional programming language: at the top level, terms to be run must be typed in an empty historical context; at runtime, historical variables get their values by substitution.

Rather than giving a specific set of ad-hoc rules for manipulating values from the historical context, we parameterize the λ^{ST} calculus over an arbitrary language with terms M, typing judgment $\Omega \vdash M : A$, and big-step semantics $M \downarrow v$. We call any such fixed choice of language the *history language*. Programs from the history language can be embedded in λ^{ST} programs using the T-HistPgm rule,

$$\frac{\Omega \vdash M : \langle s \rangle}{\Omega \mid \Gamma \vdash \langle M : s \rangle : s}\ \textsc{T-HistPgm}$$

[7] Cyclic proof systems usually ensure soundness by imposing a guardedness condition [15] which requires certain rules be applied before a back-edge can be inserted in the derivation tree. Because we are not primarily concerned with λ^{ST} as a logic at the moment, we leave a guardedness condition to future work.

which says that a historical program M of type $\Omega \vdash M : \langle s \rangle$ with access the historical context can be used in place of a λ^{ST} term of type s. Operationally, as soon as any prefix of the input arrives, we run the historical program to completion and yield the result as its stream output (after converting it into a value of type s).

How does information get added to the historical context? Intuitively, a variable in Γ (a stream that will arrive in the future) can be moved to Ω, where streams that have arrived in the past are saved, by waiting for the future to become the past! Formally, we define an operation called "wait," which allows the programmer to specify part of the incoming context and block this subcomputation until that part of the input stream has arrived in full. Once it has, we can bind it to the variables in the historical context and continue by running e.

The T-Wait-Surf rule encodes the typing content of this behavior. It allows us to specify a variable x of the input, flatten its type, and then move it to the historical context, so that the continuation e can refer to it in historical terms. Semantically, this works by buffering in environments until a maximal prefix for x has arrived. Once we have a full prefix for x, we substitute it into e and continue running the resulting term.[8] This buffering is implemented the same way as in the left rules for plus and star, by generalizing the typing rule T-Wait-Surf to a rule T-Wait which includes an explicit prefix buffer. As with plus and star, the generalized rule simplifies to the surface rule when the buffer is empty. The generalized rule and the semantics of both the wait and historical program constructs can be found in the extended version [22]. The remaining typing rules in λ^{ST} change only by adding an Ω to the typing judgment everywhere.

$$\frac{\Omega, x : \langle s \rangle \mid \Gamma(\cdot) \vdash e : s}{\Omega \mid \Gamma(x : s) \vdash \mathtt{wait}_s(x)\,(e) : s}\ \textsc{T-Wait-Surf}$$

Updated Soundness Theorems. Adding recursion and the historical context requires us to update to the soundness theorem from that of Kernel λ^{ST} to Full λ^{ST}. If a well-typed term has (a) closed historical context, and (b) no unbound recursive calls, takes a step on a well-typed input *using some amount of gas*, then the output and resulting term are also well typed.

Theorem 4.1 (Soundness of the λ^{ST} Semantics). *If* $\cdot \mid \Gamma \vdash e : s$, *and* $\eta : \mathtt{env}(\Gamma)$, *and* $\eta \Rightarrow e \downarrow^n e' \Rightarrow p$, *then* $p : \mathtt{prefix}(s)$ *and* $\cdot \mid \delta_\eta(\Gamma) \vdash e' : \delta_p(s)$

A similarly updated statement of the homomorphism theorem can be found in the full version [22].

5 DELTA

We next show how λ^{ST} addresses the problems that we identified in Section 2 of (a) type-safe programming with temporal patterns and (b) deterministic processing of parallel data. We also show how some other characteristic streaming idioms can be expressed elegantly in λ^{ST}.

The examples in this section are written in Delta,[9] an experimental language design based on λ^{ST}. Delta proposes a high-level functional syntax that, after typechecking, is desugars to λ^{ST} terms. It further supports some features that are not included in the λ^{ST} calculus, but that we expect will be required in full-blown language designs based on λ^{ST}.

5.1 Delta Syntax and Features

While the proof terms of λ^{ST} allow elimination forms (such as `let (x,y) = z in e`) to be applied only to variables (an artifact of the sequent calculus-style formalism), Delta's syntax is a standard

[8] The semantics of the T-Wait rule is reminiscent of the "blocking reads" of Kahn Process Networks, where every read from a parallel stream blocks all other reads to ensure determinism. Here, we choose a variable and block the rest of the program until it is complete and in memory.

[9] http://www.github.com/alpha-convert/delta

("natural deduction style") one where elimination forms can be applied to arbitrary expressions. Delta also includes more types than λ^{ST}, adding base types `Int` and `Bool`.

Functions and Macros. Top-level functions in Delta are simply open terms: a function definition `fun f(x : Int*) : Int* = e` elaborates and typechecks to a core term e which satisfies the typing judgment $x : \texttt{Int}^\star \vdash \texttt{e} : \texttt{Int}^\star$. Higher order functions in Delta are implemented as *macros*. A function written as `fun g<f : Int -> Int>(x : Int*) : Int* = e` is a macro which takes another function `f : Int* -> Int*` as a parameter. Calls to `g` in other functions then look like `g<f'>`, where `f'` is either (a) another function defined at top level, or (b) a call to yet another macro. If the macro `g` is recursive, its recursive calls do not receive a macro argument—all recursive usages of a macro get passed the initial macro parameter `f`. This discipline ensures that the macro usage does not depend on runtime data, and so higher-order functions can be fully resolved to λ^{ST} terms statically.

Neither of these features—standard top-level functions and higher-order macros—require the use of first-class function types, which λ^{ST} does not currently support. Defining true higher-order functions would allow for streams of *functions*, such as $(s \to t)^\star$. We hope to investigate these in future work; see Section 7.

Functions in Delta can also be (prenex-) polymorphic [61]. Polymorphic functions definitions are annotated with an list of their type arguments, like `fun f[s,t](x : s*) : t* = e`. When such a function is called, the type arguments must be passed explicitly like `f[Int,Bool]`.

Historical Arguments and Generalized Wait. Functions in Delta also take arguments for their historical contexts: a function `fun f{acc : Int}(xs : Bool*) : Int* = e` takes an in-memory `Int` argument and elaborates to a core term that satisfies the typing judgment $\texttt{acc} : \texttt{Int} \mid \texttt{xs} : \texttt{Bool}^\star \vdash e : \texttt{Int}^\star$. When `f` is called, the `acc` argument must be passed a historical program. For example, if `u : Int` is in the current historical context (and `ys : Bool*` in the regular one), `f{u + 1}(ys)` is an acceptable call to `f`.

The `wait` construct is also slightly more general in Delta. Instead of just `wait`ing on variables, programmers may `wait` on the result of some expression, and then save its result into memory. This is written `wait e as x do e' end`.

Delta Implementation. The implementation first lowers the surface syntax to an "elaborated syntax" via a transformation which eliminates shadowing, resolves function calls, and transforms the syntax into the sequent calculus representation by introducing intermediate variables for subexpressions. Elaborated terms are then typechecked. Typechecking expands macros and produces *monomorphizers* of λ^{ST} terms: functions from closed types (to plug in for type variables) to monomorphic λ^{ST} terms. Terms of base type can then be evaluated with a definitional interpreter that implements the λ^{ST} semantics.

5.2 Examples

Besides its type system, Delta's design differs from that of most stream processing languages in another important respect. In languages like Flink [30], Beam[35], and Spark [33], streaming programs must be written using a handful of provided combinators like `map`, `filter`, and `fold` (or possibly as SQL-style queries, in languages derived from CQL [6]). By contrast, Delta programs are written in the style of functional list processors. Instead of working to cram complex program behaviors into `map`s, `filter`s, and `fold`s, programmers can express their intent more directly in the form of general recursive functional programs. Of course, this does not preclude the use of the aforementioned combinators, which are directly implementable in Delta. For space, we omit the

details of many examples: Delta code for all examples can be found in the full version [22] or the Delta source repository (http://www.github.com/alpha-convert/delta).

Map. Given a transformer from `s` to `t`, we can lift it to a transformer from `s*` to `t*` with a `map` operation. The code for this function is essentially identical to the familiar functional program, but its type is more general than the standard `map` function on homogeneous streams, which has type `(a → b) → (Stream a → Stream b)`: the types s and t here can be arbitrary stream types, not just singletons.

```
fun map [s,t] <f : s -> t> (xs : s*) : t* =
  case xs of
    nil => nil
  | y :: ys => f(y) :: map(ys)
```

Filter. Similarly, given a "predicate" function f from s to $t + \varepsilon$ (the streaming version of t `option`), we can transform an incoming stream of $s^\star$ to include just the transformed elements which pass the filter.

We can then recover a traditional predicate-based filter by lifting a predicate `f` that takes an in-memory `s` to `Bool` to a streaming function `s -> s + Eps` with `liftP`. This program simply waits for its argument to arrive, then applies the predicate to the in-memory `s`.

```
fun mapMaybe[s,t]<f : s -> t + Eps> (xs : s*) : t*=
  case xs of
    nil => nil
  | y :: ys => case f y of
                 | inl(t) => t :: mapMaybe(ys)
                 | inr(_) => mapMaybe(ys)

fun liftP[s]<f : {s}(Eps) -> Bool>(x : s) : s + Eps =
  wait x, f{x}(sink) as b do
      if {b} then inl({x}) else inr(sink)
  end

fun filter<f : {s}(Eps) -> Bool>(xs : s*) : s* =
  mapMaybe[s,s]<liftPred<f>>(xs)
```

Fold. Delta can express both *running folds*, which output a stream of all their intermediate states, and *functional folds*, which output only the final state.

The (functional) `fold` transducer maintains an in-memory accumulator of type t; this gets updated by a streaming step function `f : {t}(s) -> t` that takes the state t and the new element s and produces a t. The whole fold takes a stream `xs` of type $s^\star$ and an initial accumulator value $y : t$, and it eventually produces the final state t. Folds that return only the final state cannot be given this rich type in traditional stream processing languages (for the same reason as the `head` and `tail` functions). As for `map`, the code for `fold` is very similar to the traditional functional program: the only distinction is the inclusion of `wait`s to marshal data into memory.

```
fun fold [s,t] <f : {t}(s) -> t>{acc : t}(xs : s*) : t =
  case xs of
    nil => {acc}
  | y :: ys => wait f{acc}(y) as acc' do
                 fold{acc'}(ys)
               end
```

Singletons, Head, Tail. In the homogeneous model, stream types are always conceptually unbounded. But in many practical situations, a stream will only be expected to contain a single element—a constraint that cannot be expressed with homogeneous streams. Using stream types, we can write stream transformers that are statically known to only produce a single output. For example, the "head" function is trivially expressible in the same manner as head on lists, as shown on the right.

```
fun head [s] (xs : s*) : Eps + s =
  case xs of
    nil => inl(sink)
  | y :: _ => inr(y)
```

Brightness Levels. The temporal invariant from the brightness-levels example in Section 2 can be encoded as the type $(\texttt{Int} \cdot \texttt{Int}^\star)^\star$: a stream of nonempty streams of `Int`s, representing "runs" of light levels greater than some threshold. The thresholding operation `thresh` takes a stream of `Int`s and produces runs of elements above the threshold. Whenever the incoming stream goes above the threshold `t`, we collect all of the subsequent elements into a run, emit it, and recurse down the rest of the stream. This uses an operation `spanGt : {Int} (Int*) -> Int . Int*` that

returns the initial "span" of elements above `t`, followed by the rest of the stream. It's important to note that `thresh` does not wait for a complete run to produce output: as soon as the first element above `t` arrives, it is forwarded along, as are all subsequent elements until the stream drops below `t`. By contrast with homogeneously typed streaming languages, Delta's type safety guarantees that `thresh` *does in fact* output a stream that adheres to the protocol, and ensures downstream transformers do not have to replicate this parsing logic.

To continue the example from Section 2, we can use this parsed stream of runs to compute per-run averages, by `map`ing an `averageSingle` operation—taking `Int · Int*` to `Int`—over the stream of runs. This operation is defined by computing the sum and length of a run in parallel, waiting for the results, then computing the average. If it consumed a homogeneous stream type like `(Start + Int + End)*`, this average-each-run operation would need to be written in a low-level, more stateful manner, remembering the current run of `Int`s until an `End` event arrives, averaging, and handling the divide-by-zero error which could in principle occur if no `Int`s arrived between a `Start` and an `End`. The complete program, first calling `thresh`, and then mapping `averageSingle` over the stream of runs, is `averageAbove`.

```
fun thresh{t : Int}(xs : Int*) : (Int . Int*)* =
  case xs of
    nil => nil
  | y :: ys => wait y do
                 if {y > t} then
                   let (run;rest) = spanGt{t}(ys) in
                   ({y};run) :: thresh{t}(rest)
                 else
                   thresh{t}(ys)
               end

fun averageSingle (run : Int . Int*) : Int =
  let (x;xs) = run in
  let (sm,len) = (sum(xs), length(xs)) in
  wait x,sm,len do
    {(x + sm) / (1 + len)}
  end

fun averageAbove{t : Int}(xs : Int*) : Int* =
  map<averageSingle>(thresh{t}(xs))
```

Partitioning and Merging. Partitioning is a crucial streaming idiom where a single stream of data is split into two or more parallel streams to be routed to different downstream processing nodes, thus exposing parallelism and increasing potential throughput. Examples of partitioning strategies implementable in Delta are *round-robin partitioning*—which partitions an incoming stream of type $s^\star$ into a parallel pair of streams $s^\star \| s^\star$ by sending the first s to the left, the second to the right, and so on—and *decision-based partitioning*—which routes a stream of type $s^\star$ into an output stream of type $t^\star \| r^\star$ based on the result of a function from s to $t + r$.

Dual to partitioning is *merging*, which transforms a parallel pair of streams $s^\star \| t^\star$ to a stream of parallel pairs $(s \| t)^\star$ by pairing off the first s with the first t, the second s with the second t, and so on. This operation is necessarily deterministic by Theorem 3.3, and so it prevents the bug when averaging data from a pair of sensors in Section 2.

Windowing and Punctuation. Windowing is another core concept in stream processing systems, where aggregation operations like moving averages or sums are defined over "windows"—groupings of consecutive events, gathered together into a set. In Delta, these transformers are just `map`s over a stream whose elements are windows. Given a per-window aggregation transformer `f` from an individual window $s^\star$ to a result type t, plus a "windowing strategy" `win` which takes a stream $r^\star$ and turns it into a stream of windows $s^{\star\star}$, we can write the windowed operation as `map<f>(win(xs))`. Delta can express a variety of windowing strategies, including sliding and tumbling size-based window operators, as well as punctuation-based windowing, where windows are delimited by punctuation marks inserted into the stream.

6 RELATED WORK

Streams as a programming abstraction have their sources in early work in the programming languages [17, 49, 71, 73] and database [1, 2, 5–7, 21, 57] communities. Though streams have mostly been viewed as homogeneous sequences, more interesting treatments have also been proposed. For example, streams in the database literature are sometimes viewed as time-varying relations, while the PL community has produced formalisms like process calculi and functional reactive programming. To our knowledge, ours is first type system for stream programming capturing both (1) heterogeneous patterns of events over time and (2) combinations of parallel and sequential data.

Sequential, homogeneous streams and dataflow programs. Traditionally, streams have been viewed in the PL community as coinductive sequences [17]: a stream of `A` has a single (co)constructor, $\texttt{cocons} : \texttt{Stream}A \to (A \times \texttt{Stream}\ A)$ and acts as a lazily evaluated infinite list. In particular, this is the setting of traditional *dataflow programming* [71]. One major challenge in reasoning about dataflow over sequential streams is the nondeterminism arising from operators whose output may depend on the order in which events arrive on multiple input streams. Kahn's seminal "process networks" [49] (including their restriction to synchronous networks [12, 56, 73]) avoid this problem by allowing only *blocking reads* of messages on FIFO queues. In contrast, the semantics of λ^{ST} leverages its type structure to guarantee deterministic parallel processing *without* blocking in many cases. For example, in the context of a T-Let rule, if the type system can detect statically that a transformer is using two parallel streams safely, it can read from them simultaneously.

Partitioned streams. Building on streams as homogeneous sequences, modern stream processing systems such as Flink [18, 30], Spark Streaming [33, 78], Samza [32, 62], Arc [55], and Storm [34] support *dynamic partitioning*: a stream type can define one stream with many parallel substreams, where the number of substreams and assignment of data to substreams is determined at runtime. The type `Stream t` in these systems is implicitly a parallel composition of homogeneous streams: $\texttt{t}^\star \| \cdots \| \texttt{t}^\star$. Unlike in λ^{ST}, these parallel substreams cannot have more general types.

Some which papers which attempt to build very general compile targets for stream processing support parallelism in only restricted ways. For example, Brooklet [69] and the DON Calculus [25] support data parallelism only as an optimization pass in limited cases. This is because stream partitioning does not in general preserve the semantics of the source program and can introduce undesirable nondeterminism [42, 59, 67]. While λ^{ST} does not support dynamic partitioning, we hope to address it in future work; see Section 7.

Streams as time-varying relations. In the database literature, streams are often viewed as relations (sets of tuples) that vary over time. Stream management systems in the early 2000s pioneered this paradigm, including Aurora [2] and Borealis [1], TelegraphCQ [21] and CACQ [57], and STREAM [5]. A time-varying relation can be viewed as either a function from timestamps to finite relations or an infinite set of timestamped values; this correspondence was elegantly exploited by early streaming query languages such as CQL [6, 7] and remains popular today [11, 46]. Time-varying relations can be expressed in λ^{ST} using Kleene star and concatenation: a relation of tuples of type `T` timestamped by `Time` can be expressed as $(\texttt{T}^\star \cdot \texttt{Time})^\star$. We can also express the common pattern where parallel streams are synchronized by a single timestamp (again, modulo dynamic partitioning) with types like $\left((\texttt{T}^\star \| \texttt{T}^\star) \cdot \texttt{Time}\right)^\star$. Each `Time` event is a punctuation mark containing the timestamp of the prior set of tuples [48, 76]. Traditional systems include separate APIs for operations that modify punctuation (e.g., a *delay* function that increments timestamps); whereas in our system they are ordinary stream operators and punctuation markers are ordinary events.

Streams as Pomsets and Monoids. A sweet spot between the homogeneous sequential and relational viewpoints is found in prior work treating streams as *pomsets* (partially ordered multisets) [3, 50–52, 59], inspired by work in concurrency theory [26, 60]. In a pomset, data items may be completely

ordered (a sequence), completely unordered (a bag), or somewhere in between. Some recent works have proposed pomset-based and structured monoid-based types for streams [3, 58, 59], but their types do not include concatenation and do not come with type *systems*—programs must be shown to be well typed semantically, rather than via syntactic typing rules.

Functional reactive programming (FRP) [27] treats programs as incremental, reactive state machines written using functional combinators. The fundamental abstraction is a "signal": a time-varying value `Sig(A) = Time -> A`. Work on type systems for FRP has used modal and substructural types [8, 9, 19, 54] to guarantee properties like causality, productivity, and space leak freedom. While our type system is not *designed* to address these issues, it does incidentally have bearing on them. For one, our incremental semantics demonstrates that λ^{ST}'s type system enforces causality: since outputs that have been incrementally emitted cannot be retracted or changed, the type system must ensure that past outputs cannot depend on future inputs. Similarly, potential space leaks can be detected statically by checking that only bounded-sized types are buffered using `wait` or the buffering built into the left rules for sums and star. Our current calculus does not guarantee productivity (new inputs must eventually produce new outputs), but in Section 7 we discuss how to remedy this by imposing guardedness conditions on recursive calls.

Jeffrey [47] permits the type of a signal to vary over time, using dependent types inspired by Linear Temporal Logic [65]. This system includes an *until* type that behaves like our concatenation type: a signal of type $A\,U\,B$ is a signal of type A, followed by a signal of type B. However, unlike parallel streams in our setting, time updates in steps, discretely; i.e., parallel signals all present new values together, at the same time. Concurrently with our work, Bahr and Møgelberg [10] proposes a modal type system to weaken the synchronicity assumption; however, it still treats signals as homogeneous: the type of data cannot change over time. Lastly, Paykin et al. [64] develop a modal type system which expresses low-level event handlers. These are also purely synchronous, and the programs are written as event handlers as opposed to high-level "batch" processors.

Stream Runtime Verification (SRV) aims, broadly, to monitor streams at runtime and provide boolean or numerical "triggers" that fire when they satisfy some specification. Many RV projects like LOLA [23], HLola [20], RTLola [28], Striver [41], HStriver [40] also provide high-level, declarative specification languages for writing such monitors. Because these languages often use regular expressions or LTL as a formalism, they often bear a resemblance to our stream types. Despite this similarity, our goals and methods are quite different. Unlike the dynamically-checked specifications of SRV, the types in Delta are static guarantees: a stream program of type s necessarily produces a stream of type s.

Streaming with Laziness. It is folklore in the Haskell community that a "sufficiently lazy" list program can be run as a streaming program using a clever trick with lazy IO [53, 74]. This "sufficient laziness" condition is syntactically brittle, and requires an expert Haskell programmer to carefully ensure that all functions involved are lazy in the just the right way. Indeed, many Haskell programmers instead reach for combinator libraries like Pipes [38], FoldL [39], Conduit [68], Streamly [72], and others to ensure their programs have a streaming semantics. In Delta, the type system takes care of this for you: all well-typed programs can be given a streaming semantics. Moreover, the λ^{ST} semantics gives a direct account of how pure functions execute incrementally as state machines, as opposed to the way that Haskell's non-strict semantics incidentally yields streaming behavior when combined with Lazy IO.

Session types and process calculi. Another large body of work with similar vision is session types for process calculi [44], where types describe complex sequential protocols between communicating processes as they evolve through time. A main difference from our work is that the session type of a process describes the *protocol* for its communications with other processes—i.e., the sequence of sends and receives on different channels—while the stream type of a λ^{ST} program describes

only the data that it communicates. Indeed, a stream transformer might display many patterns of communication with downstream transformers: it can run in "batch mode"—sending exactly one output after accepting all available input—or in a sequence smaller steps, sending along partial outputs as it receives partial inputs. Also, a single channel in a process calculus cannot carry parallel substreams: all events in a channel are ordered relative to each other. Recently, Frumin et al. [37] proposed a session-types interpretation of BI that uses the bunched structure very differently from λ^{ST}. In particular, processes of type $A * B$ and $A \wedge B$ both behave semantically like a process of type A in parallel with a process of type B, while, in λ^{ST}, $s \cdot t$ and $s \| t$ describe very different streams.

Concurrent Kleene Algebras and regular expression types. Stream types are partly inspired by Concurrent Kleene Algebras (CKAs) [43] and related syntaxes for pomset languages [52], but we are apparently the first to use these formalisms as *types* in a programming language rather than as a tool for reasoning about concurrency. In particular, traditional applications of Kleene algebra such as NetKAT [4] and Concurrent NetKAT [77] use KA to model *programs*, whereas in λ^{ST} we use the KA structure to describe the *data* that programs exchange, while the programs themselves are written in a separate language. We have also taken inspiration from languages for programming with XML data [13, 36, 45, etc.] using types based on regular expressions.

7 CONCLUSIONS AND FUTURE WORK

We have proposed a new static type system for stream programming, motivated by a novel variant of BI logic and able to capture both complex temporal patterns and deterministic parallel processing.

In the future, we hope to add more types to λ^{ST}. Adding a support for *b*ags—unbounded parallelism, the parallel analog of Kleene star—would enable dynamic partitioning. λ^{ST} also lacks function types. The proof theory of BI would imply that there should be two (one for each context former), but we have yet to investigate what these functions might mean in the streaming setting.

Further theoretical investigations include (1) alternate semantics for stream types, including a denotational semantics as pomset morphisms, Kahn Process Networks [49], or some category of state machines, (2) eliminating the inertness restriction on let-bindings, and (3) adding a *guardedness* condition on recursive calls to ensure termination and hence productivity.

On the applied side, we plan to build a distributed implementation of Delta by compiling λ^{ST} terms to programs for an existing stream processing system like Apache Storm [34], thus inheriting its desirable fault-tolerance and delivery guarantees. We hope to build such a compiler and use it as a platform for experimenting with type-enabled optimizations and resource usage analysis.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. We also thank Justin Lubin for feedback on drafts of this paper, Will Sturgeon for help with formalizing the most intricate details of the theory of stream types, and PLClub, Alex Kavvos, Andrew Hirsch, Mae Milano, and Michael Arntzenius for helpful discussions about this work. Cutler was supported by a NSF Graduate Research Fellowship under grant number 2022334433, Waston by NSF awards 1763514 and 2331783, Hilliard by NSF Award III-1910108, and Pierce and Goldstein by NSF Award 1421243, *Random Testing for Language Design*.

REFERENCES

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley Zdonik. 2005. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR)*.

[2] Daniel J Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal* 12, 2 (2003). https://doi.org/10.1007/s00778-003-0095-z

[3] Rajeev Alur, Phillip Hilliard, Zachary G Ives, Konstantinos Kallas, Konstantinos Mamouras, Filip Niksic, Caleb Stanford, Val Tannen, and Anton Xue. 2021. Synchronization Schemas. *Invited contribution, Principles of Database Systems.*

[4] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic foundations for networks. *Acm sigplan notices* 49, 1 (2014), 113–126.

[5] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. 2004. *STREAM: The Stanford Data Stream Management System.* Technical Report 2004-20. Stanford InfoLab.

[6] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2003. CQL: A language for continuous queries over streams and relations. In *International Workshop on Database Programming Languages*. Springer.

[7] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* 15, 2 (2006). https://doi.org/10.1007/s00778-004-0147-z

[8] Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2019. Simply RaTT: A Fitch-Style Modal Calculus for Reactive Programming without Space Leaks. *Proc. ACM Program. Lang.* 3, ICFP, Article 109 (jul 2019), 27 pages. https://doi.org/10.1145/3341713

[9] Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2021. Diamonds Are Not Forever: Liveness in Reactive Programming with Guarded Recursion. *Proc. ACM Program. Lang.* 5, POPL, Article 2 (jan 2021), 28 pages. https://doi.org/10.1145/3434283

[10] Patrick Bahr and Rasmus Ejlers Møgelberg. 2023. Asynchronous Modal FRP. arXiv:2303.03170 [cs.PL]

[11] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. 2019. One SQL to Rule Them All-an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables. In *International Conference on Management of Data (SIGMOD)*.

[12] Albert Benveniste, Paul Caspi, Stephen A Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (2003).

[13] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. 2003. CDuce: An XML-Centric General-Purpose Language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 51–63.

[14] Gérard Berry and Georges Gonthier. 1992. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming* 19, 2 (1992), 87–152. https://doi.org/10.1016/0167-6423(92)90005-V

[15] James Brotherston. 2005. Cyclic Proofs for First-Order Logic with Inductive Definitions. In *Automated Reasoning with Analytic Tableaux and Related Methods (Lecture Notes in Computer Science)*, Bernhard Beckert (Ed.). Springer, Berlin, Heidelberg, 78–92. https://doi.org/10.1007/11554554_8

[16] Janusz A Brzozowski. 1964. Derivatives of regular expressions. *J. ACM* 11, 4 (1964).

[17] William H Burge. 1975. Stream processing functions. *IBM Journal of Research and Development* 19, 1 (1975).

[18] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[19] Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. 2014. Fair Reactive Programming. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 361–372. https://doi.org/10.1145/2535838.2535881

[20] Martín Ceresa, Felipe Gorostiaga, and César Sáchez. 2020. Declarative Stream Runtime Verification (hLola). In *Proc. of the 18th Asian Symposium on Programming Languages and Systems (APLAS'20) (LNCS, Vol. 12470)*. Springer, 25–43. https://doi.org/10.1007/978-3-030-64437-6_2

[21] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R Madden, Fred Reiss, and Mehul A Shah. 2003. TelegraphCQ: continuous dataflow processing. In *ACM SIGMOD International Conference on Management of Data (SIGMOD*. 668–668.

[22] Joseph W. Cutler, Christopher Watson, Phillip Hilliard, Harrison Goldstein, Caleb Stanford, and Benjamin C. Pierce. 2023. Stream Types. arXiv:2307.09553 [cs.PL]

[23] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H.B. Sipma, S. Mehrotra, and Z. Manna. 2005. LOLA: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*. 166–174. https://doi.org/10.1109/TIME.2005.26

[24] Farzaneh Derakhshan. 2021. *Session-Typed Recursive Processes and Circular Proofs.* Ph. D. Dissertation. Caregie Mellon University. https://www.andrew.cmu.edu/user/fderakhs/publications/Dissertation_Farzaneh.pdf

[25] Philip Dexter, Yu David Liu, and Kenneth Chiu. 2022. The essence of online data processing. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 899–928.

[26] Volker Diekert and Grzegorz Rozenberg. 1995. *The Book of Traces.* World Scientific. https://doi.org/10.1142/2563

[27] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *Second ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

[28] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah. 2019. StreamLAB: Stream-based Monitoring of Cyber-Physical Systems. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, 421–431.
[29] Jérôme Fortier and Luigi Santocanale. 2013. Cuts for circular proofs: semantics and cut-elimination. In *Computer Science Logic 2013 (CSL 2013) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 23)*, Simona Ronchi Della Rocca (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 248–262. https://doi.org/10.4230/LIPIcs.CSL.2013.248 ISSN: 1868-8969.
[30] Apache Software Foundation. 2019. Apache Flink. https://flink.apache.org/. (Accessed July 2022.).
[31] Apache Software Foundation. 2019. Apache Heron (originally Twitter Heron). https://heron.incubator.apache.org/. (Accessed July 2022.).
[32] Apache Software Foundation. 2019. Apache Samza. https://samza.apache.org/. (Accessed July 2022.).
[33] Apache Software Foundation. 2019. Apache Spark Streaming. https://spark.apache.org/streaming/. (Accessed July 2022.).
[34] Apache Software Foundation. 2019. Apache Storm. https://storm.apache.org/. (Accessed July 2022.).
[35] Apache Software Foundation. 2021. Apache Beam. https://beam.apache.org/. (Accessed July 2022.).
[36] Alain Frisch, Giuseppe Castagna, and Veronique Benzaken. 2002. Semantic Subtyping. In *Logic in Computer Science (LICS)*.
[37] Dan Frumin, Emanuele D'Osualdo, Bas van den Heuvel, and Jorge A. Pérez. 2022. A Bunch of Sessions: A Propositions-as-Sessions Interpretation of Bunched Implications in Channel-Based Concurrency. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 155 (oct 2022), 29 pages. https://doi.org/10.1145/3563318
[38] Gabriella Gonzalez. 2022. Pipes. https://hackage.haskell.org/package/pipes.
[39] Gabriella Gonzalez. 2024. FoldL. https://hackage.haskell.org/package/foldl.
[40] Felipe Gorostiaga and César Sánchez. 2021. HStriver: A Very Functional Extensible Tool for the Runtime Verification of Real-Time Event Streams. In *Proc. of the 24th Int'l Symp. on Formal Methods (FM'21) (LNCS, Vol. 13047)*. Springer, 563–580. https://doi.org/10.1007/978-3-030-90870-6_30
[41] Felipe Gorostiaga and César Sánchez. 2021. Stream runtime verification of real-time event streams with the Striver language. *International Journal on Software Tools for Technology Transfer* 23 (2021), 157–183. https://doi.org/10.1007/s10009-021-00605-3
[42] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)* 46, 4 (2014).
[43] CAR (Tony) Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. 2009. Concurrent Kleene Algebra. In *CONCUR 2009-Concurrency Theory: 20th International Conference, CONCUR 2009, Bologna, Italy, September 1-4, 2009. Proceedings 20*. Springer, 399–414.
[44] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 273–284.
[45] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. 2005. Regular Expression Types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27, 1 (Jan. 2005), 46–90. Preliminary version in ICFP 2000.
[46] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. 2008. Towards a streaming SQL standard. *Proceedings of the VLDB Endowment* 1, 2 (2008).
[47] Alan Jeffrey. 2012. LTL Types FRP: Linear-Time Temporal Logic Propositions as Types, Proofs as Functional Reactive Programs. In *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification* (Philadelphia, Pennsylvania, USA) *(PLPV '12)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/2103776.2103783
[48] Theodore Johnson, Shanmugavelayutham Muthukrishnan, Vladislav Shkapenyuk, and Oliver Spatscheck. 2005. A heartbeat mechanism and its application in Gigascope. In *31st International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment.
[49] Gilles Kahn. 1974. The semantics of a simple language for parallel programming. *Information Processing* 74 (1974).
[50] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. 2020. DiffStream: differential output testing for stream processing programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020).
[51] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. 2022. Stream Processing With Dependency-Guided Synchronization. In *Principles and Practice of Parallel Programming (PPoPP)*.
[52] Tobias Kappé, Paul Brunet, Bas Luttik, Alexandra Silva, and Fabio Zanasi. 2019. On series-parallel pomset languages: Rationality, context-freeness and automata. *Journal of Logical and Algebraic Methods in Programming* 103 (2019), 130–153. https://doi.org/10.1016/j.jlamp.2018.12.001
[53] Oleg Kiselyov. 2012. Iteratees. In *Functional and Logic Programming*, Tom Schrijvers and Peter Thiemann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 166–181.

[54] Neelakantan R. Krishnaswami. 2013. Higher-Order Functional Reactive Programming without Spacetime Leaks. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 221–232. https://doi.org/10.1145/2500365.2500588

[55] Lars Kroll, Klas Segeljakt, Paris Carbone, Christian Schulte, and Seif Haridi. 2019. Arc: an IR for batch and stream programming. In *Proceedings of the 17th ACM SIGPLAN International Symposium on Database Programming Languages* (Phoenix, AZ, USA) *(DBPL 2019)*. Association for Computing Machinery, New York, NY, USA, 53–58. https://doi.org/10.1145/3315507.3330199

[56] Edward A Lee and David G Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987).

[57] Samuel Madden, Mehul Shah, Joseph M Hellerstein, and Vijayshankar Raman. 2002. Continuously Adaptive Continuous Queries over Streams. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 12 pages. https://doi.org/10.1145/564691.564698

[58] Konstantinos Mamouras. 2020. Semantic Foundations for Deterministic Dataflow and Stream Processing. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 394–427.

[59] Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G Ives, and Val Tannen. 2019. Data-trace types for distributed stream processing systems. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[60] Antoni Mazurkiewicz. 1986. Trace theory. In *Advanced course on Petri nets*. Springer.

[61] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

[62] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. 2017. Samza: Stateful Scalable Stream Processing at LinkedIn. *Proceedings of the VLDB Endowment* 10, 12 (2017).

[63] Peter W O'Hearn and David J Pym. 1999. The logic of bunched implications. *Bulletin of Symbolic Logic* 5, 2 (1999), 215–244.

[64] Jennifer Paykin, Neelakantan R. Krishnaswami, and Steve Zdancewic. 2016. The Essence of Event-Driven Programming. (2016).

[65] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. ieee, 46–57.

[66] John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 55–74.

[67] Scott Schneider, Martin Hirzel, Buğra Gedik, and Kun-Lung Wu. 2013. Safe data parallelism for general streaming. *IEEE Trans. Comput.* 64, 2 (2013).

[68] Michael Snoyman. 2023. Conduit. https://hackage.haskell.org/package/conduit.

[69] Robert Soulé, Martin Hirzel, Robert Grimm, Buğra Gedik, Henrique Andrade, Vibhore Kumar, and Kun-Lung Wu. 2010. A universal calculus for stream processing languages. In *European Symposium on Programming (ESOP)*. Springer.

[70] Caleb Stanford. 2022. *Safe Programming over Distributed Streams.* Ph. D. Dissertation. University of Pennsylvania.

[71] Robert Stephens. 1997. A survey of stream processing. *Acta Informatica* 34, 7 (1997).

[72] Composewell Technologies. 2023. StreamLy. https://hackage.haskell.org/package/streamly-core.

[73] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*. Springer.

[74] Jose Manuel Calderon Trilla. 2024. personal communication.

[75] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Trans. on Knowl. and Data Eng.* 15, 3 (mar 2003), 555–568. https://doi.org/10.1109/TKDE.2003.1198390

[76] Peter A Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering* 15, 3 (2003).

[77] Jana Wagemaker, Nate Foster, Tobias Kappé, Dexter Kozen, Jurriaan Rot, and Alexandra Silva. 2022. Concurrent NetKAT: Modeling and analyzing stateful, concurrent networks. In *European Symposium on Programming*. Springer International Publishing Cham, 575–602.

[78] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *24th Symposium on Operating Systems Principles (SOSP)*. ACM. https://doi.org/10.1145/2517349.2522737

Received 2023-11-16; accepted 2024-03-31